Solving Problems with Search

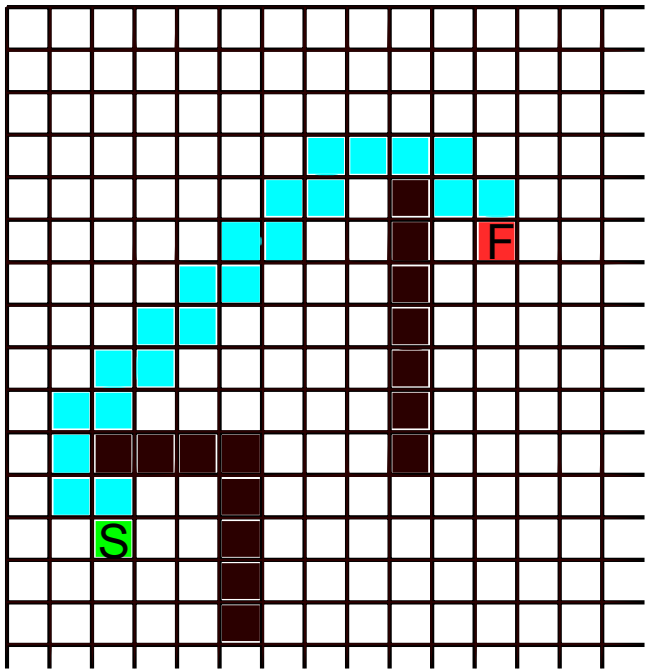
ITI0210, lecture 3 (2021)

Review

Pathfinding on 2D grid

- Expand BFS or best-first from start
- Solution is a sequence of steps

(up, left, up, …)



http://www.redblobgames.com/pathfinding/a-star/introduction.html

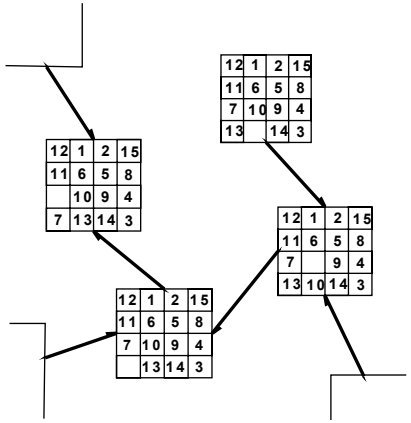
Part I: Search space

State space search

Actual problem

Physical

State space

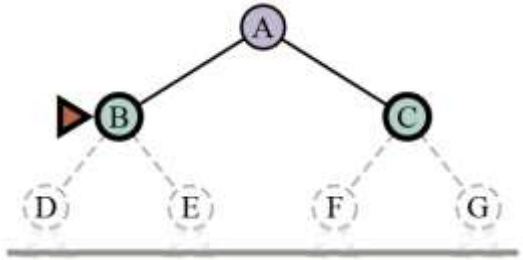


Mathematical
(may be infinite)

Representation in code

```
7
8   state = [12, 1, 2, 15, 1
9   goal = [1, 2, 3, 4, 5, 6
10
11  def actions(state):
12      empty = state.index(
13      # move up
14      up = empty - 3
15      if up >= 0:
16          yield UP_ACTION
```

Search tree



Implementation

Step 1: describe state space

Formulating a problem:

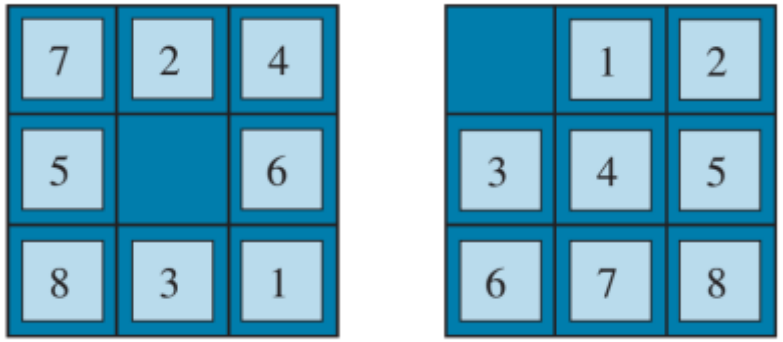
1. What are states? (grid squares)
2. State transitions? (moving left, right, up, down)
3. Where do I start? (square x=2, y=3)
4. What is the goal? (square x=11, y=9)

(blue: examples for 2D grid search)

Example formulation

The 8-Puzzle (smaller version of 15-puzzle)

Start and end states:

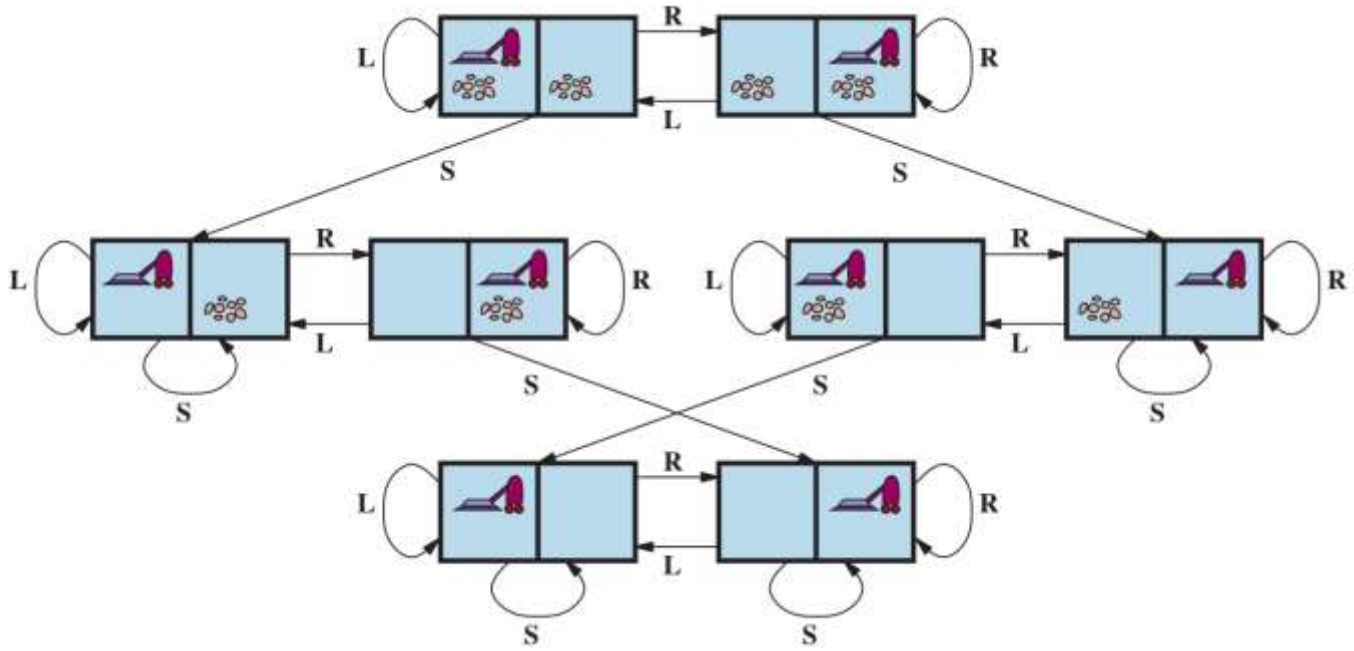


State: one possible placement of pieces

State transitions: move a piece (e.g. “3 up” in start state)

Example formulation

Complete state space for the Vacuum World



L-left, R-right, S-suck (vacuum)

Step 2: implement in code

The 8-puzzle

State representation: array

map of array index to 3x3 coordinate:

0	1	2
3	4	5
6	7	8

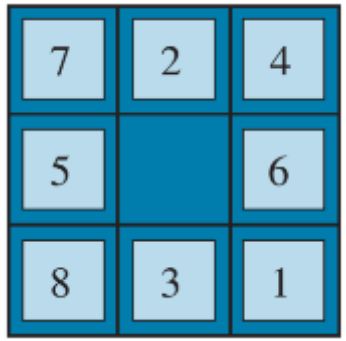
Step 2: implement in code

The 8-puzzle

State representation: array

map of 3x3 coordinate to array index:

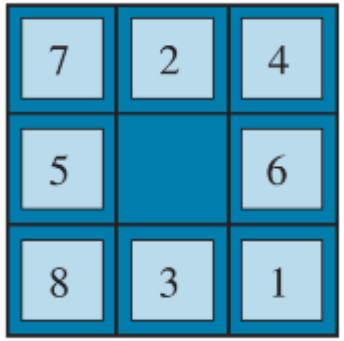
0	1	2
3	4	5
6	7	8



```
[7, 2, 4, 5, 0, 6, 8, 3, 1]
```

Step 2: implement in code

Move representation in 8-puzzle

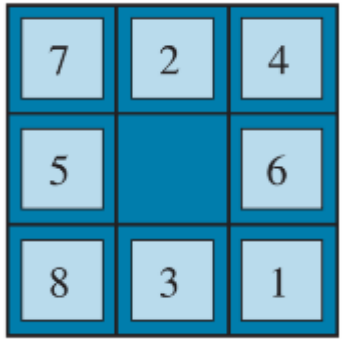


What are the possible moves?

How to find them efficiently?

Step 2: implement in code

Move representation in 8-puzzle



Idea: “move” the empty spot

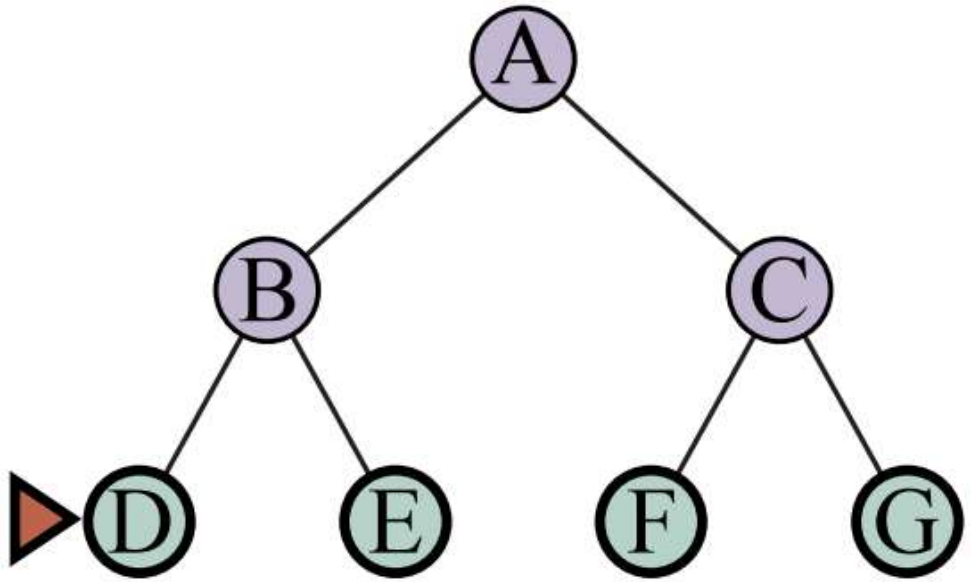
```
7
8   state = [7, 2, 4, 5, 0, 6, 8, 3, 1]
9
10  def actions(state):
11      empty = state.index(0)
12      # move up
13      up = empty - 3
14      if up >= 0:
15          # legal move, generate "UP" action
```

Exercise: what are $empty - 3$, $empty + 3$, $empty + 1$ etc?

Step 3: tree implementation

Node D needs:

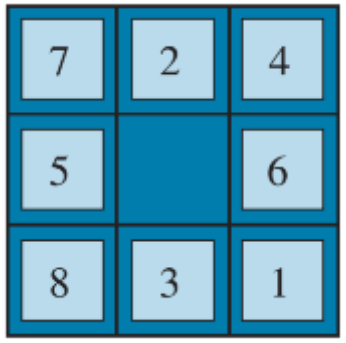
- Linked state
- Path cost (A-B-D)
- Parent (B)
- Action (from B to D)

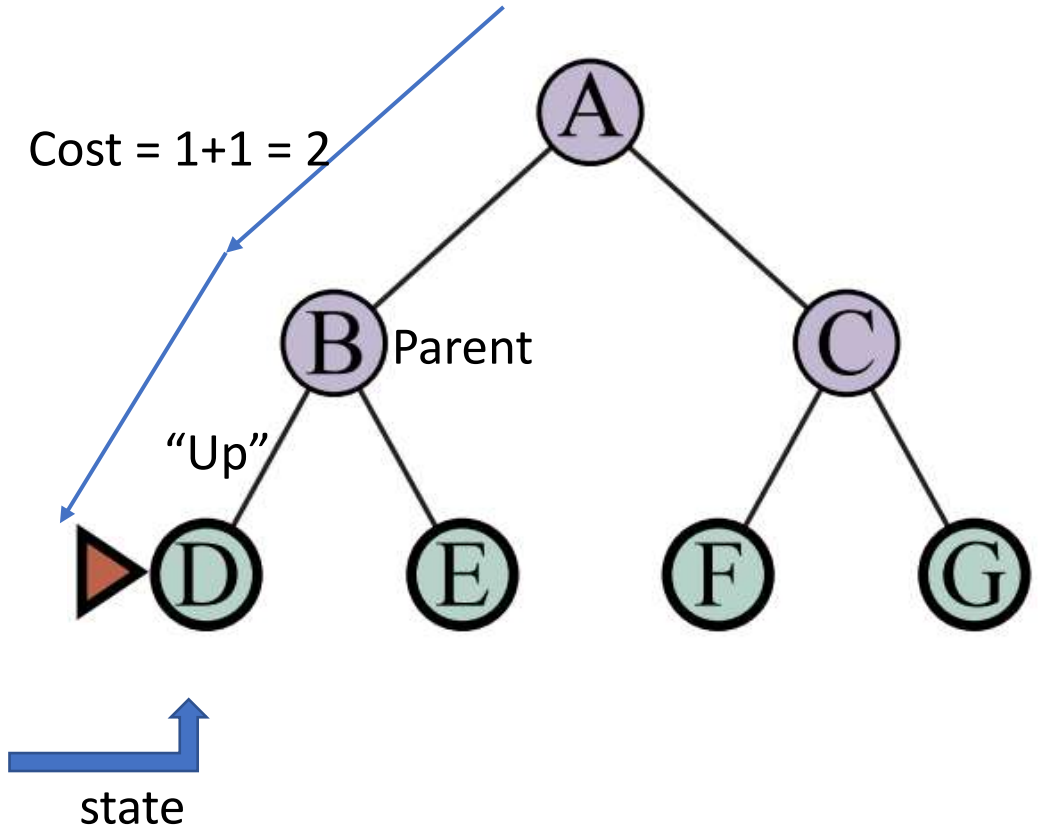


Step 3: tree implementation

Node D needs:

- Linked state
- Path cost (A-B-D)
- Parent (B)
- Action (from B to D)

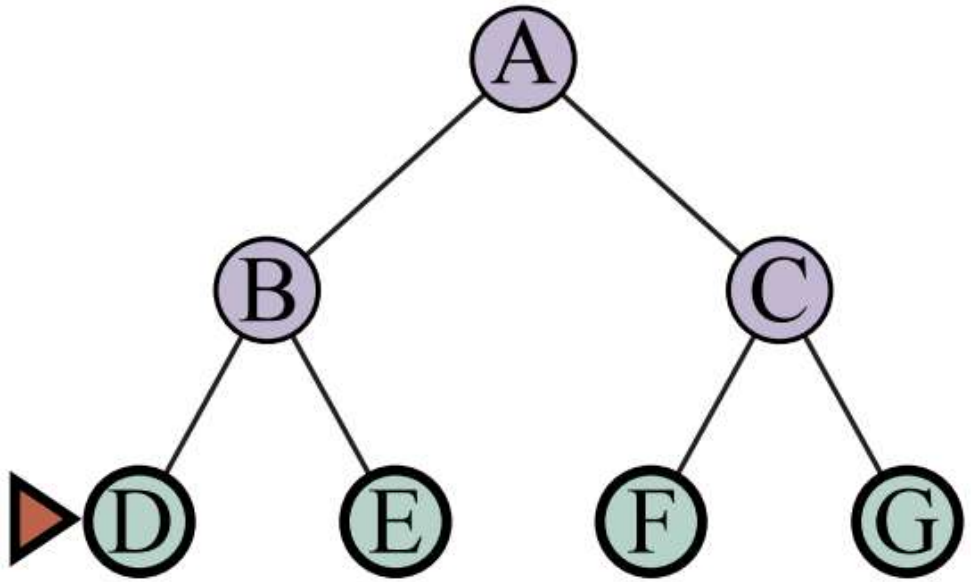




Step 3: tree implementation

Node D needs:

- Linked state
- Path cost (A-B-D)
- Parent (B)
- Action (from B to D)



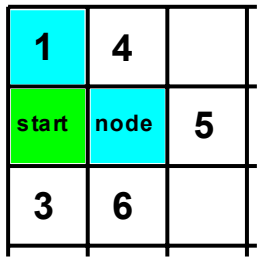
Exercise: find these things from

https://www.redblobgames.com/pathfinding/a-star/introduction.html

Part II: Search algorithm

Generic search algorithm

SearchQueue() – keep nodes ordered by strategy



goal_check() – compare to goal state or condition

expand() – generate child nodes for a node

```
def search(start):
    # start: node representing start state
    queue = SearchQueue()
    queue.add(start)
    while not queue.empty():
        node = queue.get()
        if goal_check(node):
            # solution node->parent->...->start
            return node
        for child in expand(node):
            # insertion point depends on strategy
            queue.add(child)
    return "Failed"
```

Generic search algorithm

Breadth first, A* and other algorithms:

strategies of what direction to take in search

Queue order controls the search direction

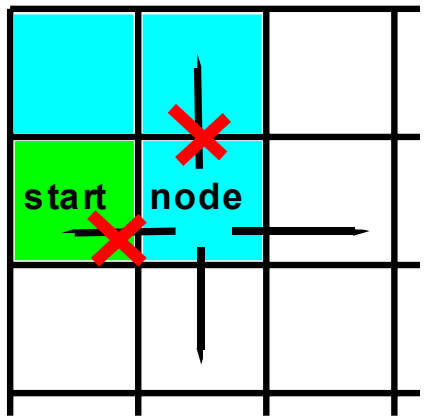
```
def search(start):
    # start: node representing start state
    queue = SearchQueue()
    queue.add(start)
    while not queue.empty():
        node = queue.get()
        if goal_check(node):
            # solution node->parent->...->start
            return node
        for child in expand(node):
            # insertion point depends on strategy
            queue.add(child)
    return "Failed"
```

Generic search algorithm

Common optimization:

Keep track of visited states,

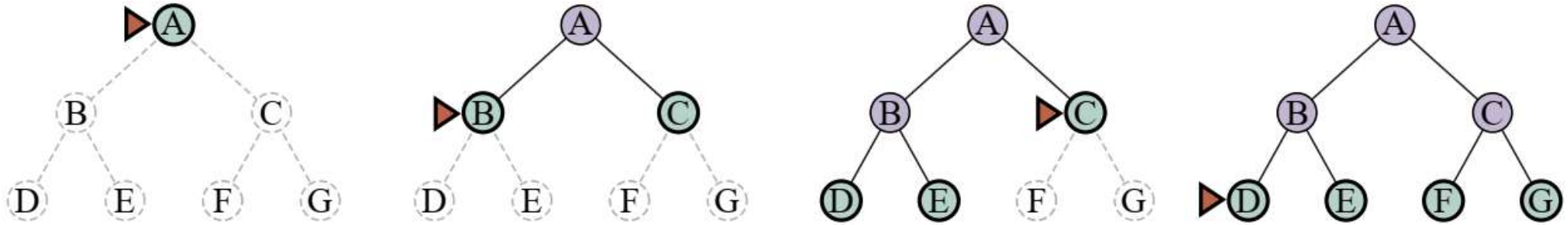
do not add them to queue




```
def search(start):
    # start: node representing start state
    queue = SearchQueue()
    queue.add(start)
    while not queue.empty():
        node = queue.get()
        if goal_check(node):
            # solution node->parent->...->start
            return node
        for child in expand(node):
            # insertion point depends on strategy
            queue.add(child)
    return "Failed"
```

Part III: Strategies

Breadth First Search



Queue:

n	**A**					
d	0					
g	0					

n	**B**	**C**				
d	1	1				
g	1	1				

n	**C**	**D**	**E**			
d	1	2	2			
g	1	2	2			

n	**D**	**E**	**F**	**G**		
d	2	2	2	2		
g	2	2	2	2		

Search step: remove 1st node and expand; new nodes to **end** of Q

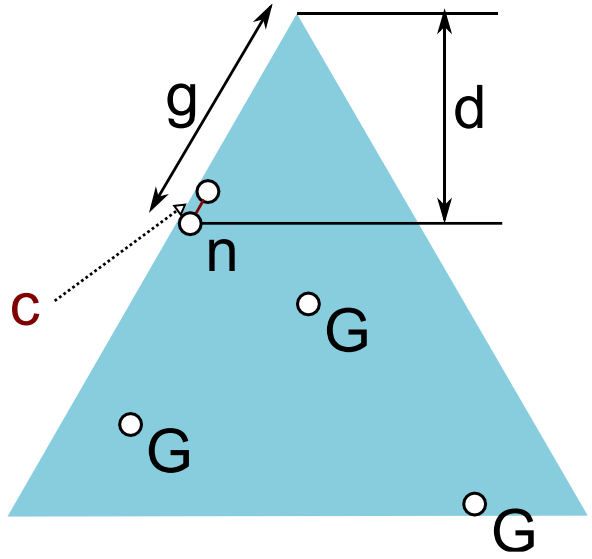
Search tree notation

n – node

d – node depth

c – step cost

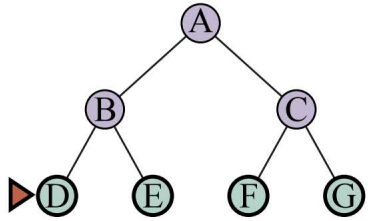
g – path cost



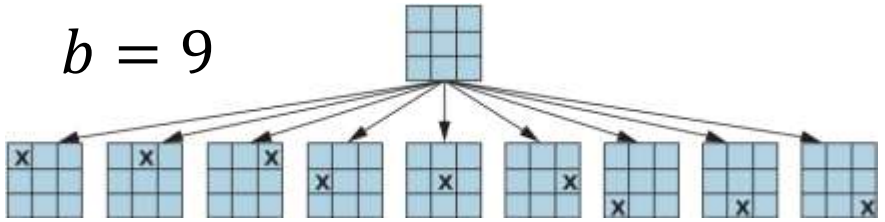
G – goal node(s)

b – branching factor

$b = 2$

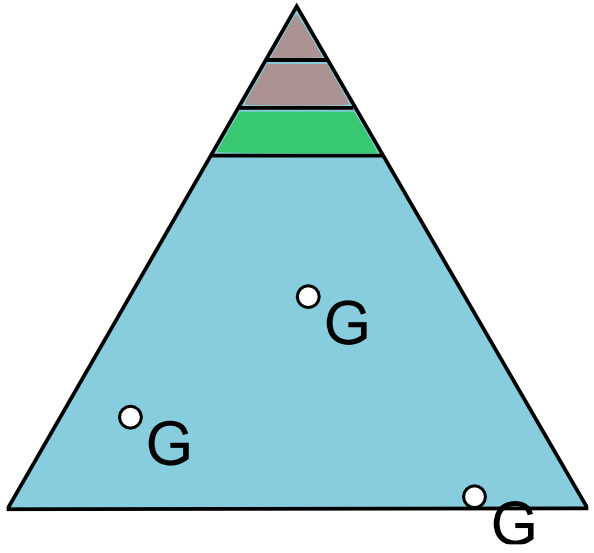


$b = 9$



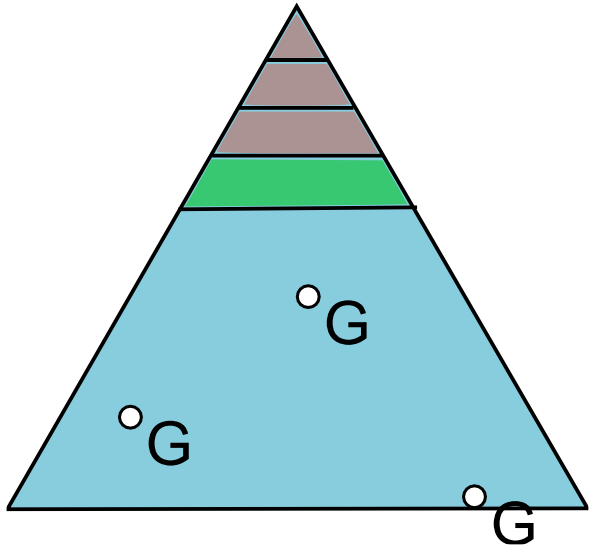
Breadth First Search

Progression



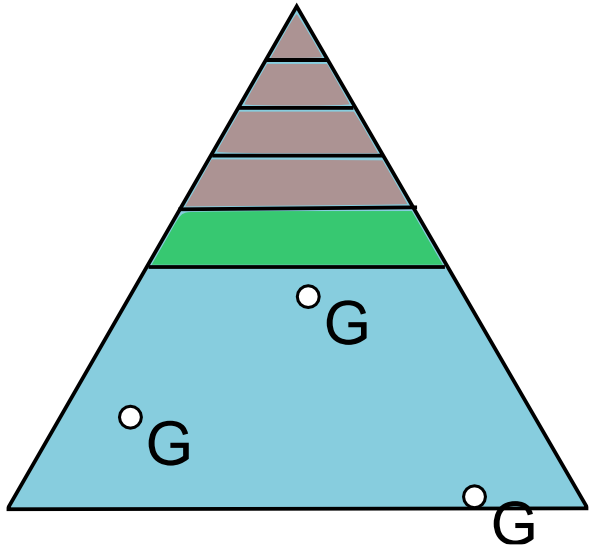
Breadth First Search

Progression



Breadth First Search

Progression



Breadth First Search

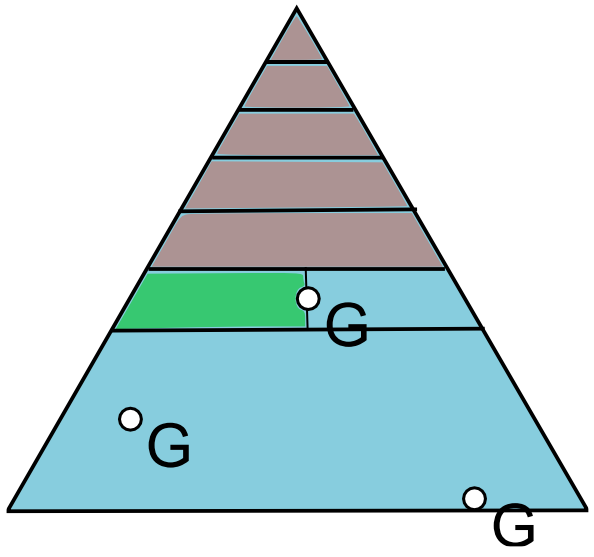
Goal found:

Minimal depth goal –

Optimal if we care about number of steps

Always finds

finite depth goal



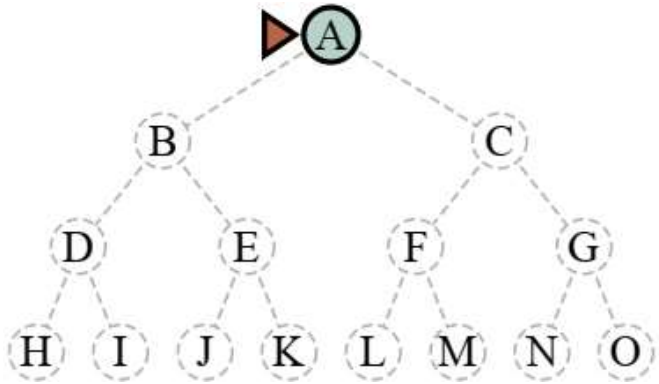
Space needed $O(b^d)$

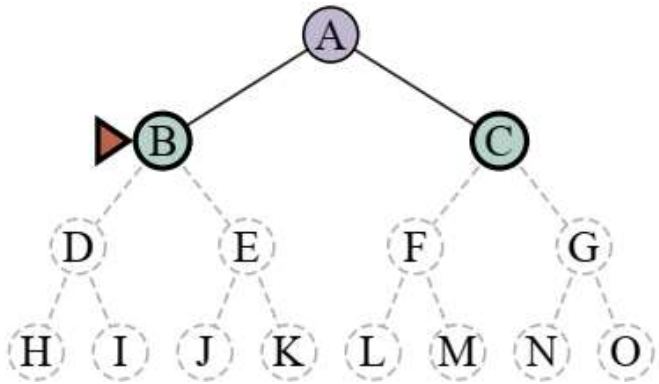
(d – goal depth)

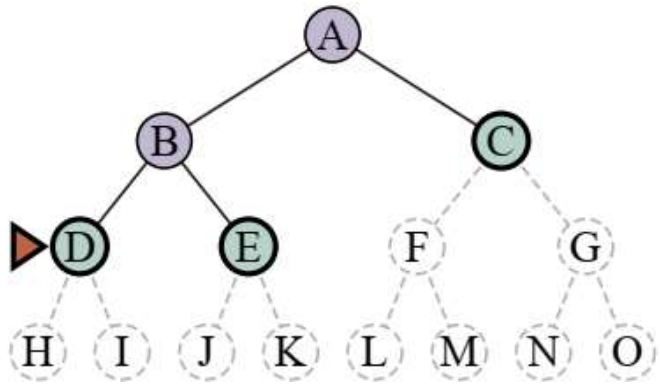
Time needed $O(b^d)$

(assuming goal check before Q insertion)

Depth First Search







Queue:

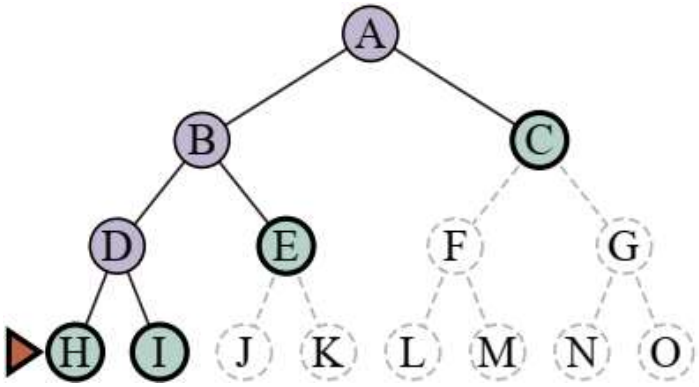
n	**A**					
d	0					
g	0					

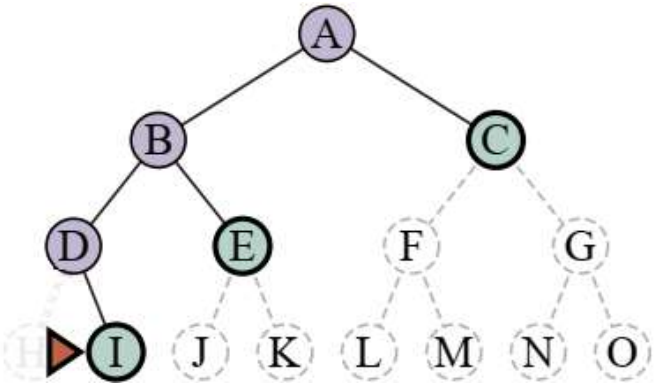
n	**B**	**C**				
d	1	1				
g	1	1				

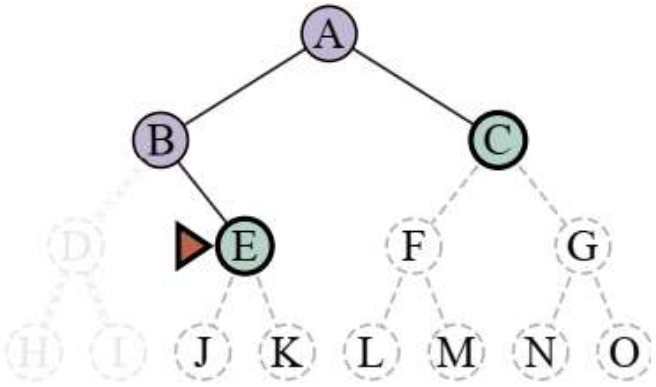
n	**D**	**E**	**C**			
d	2	2	1			
g	2	2	1			

Search step: remove 1st node and expand; new nodes to **start** of Q

Depth First Search







Queue:

n	**H**	**I**	**E**	**C**		
d	3	3	2	1		
g	3	3	2	1		

n	**I**	**E**	**C**			
d	3	2	1			
g	3	2	1			

n	**E**	**C**				
d	2	1				
g	2	1				

Search step: remove 1st node and expand; new nodes to **start** of Q

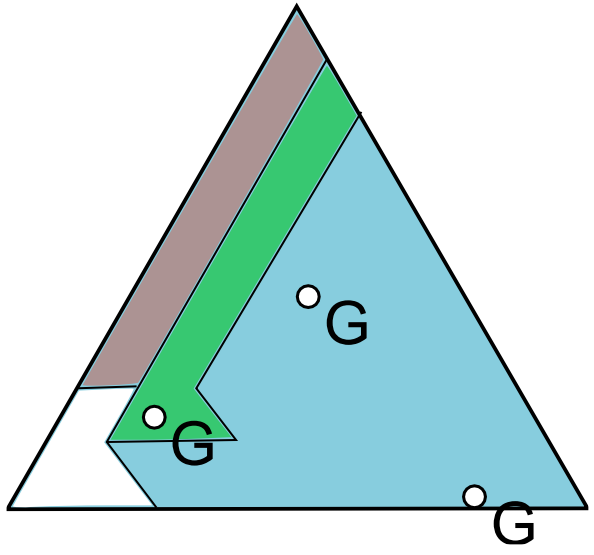
Depth First Search

Not Optimal

(may “shoot past” optimal path)

Finite depth goal

not guaranteed (∞ tree)



Space needed $O(bm)$

(m – max depth)

Time needed $O(b^m)$

Depth First Search

How to handle ∞ trees?

- Set depth limit
- Call DFS repeatedly with limit 1, 2, …

This variant is called

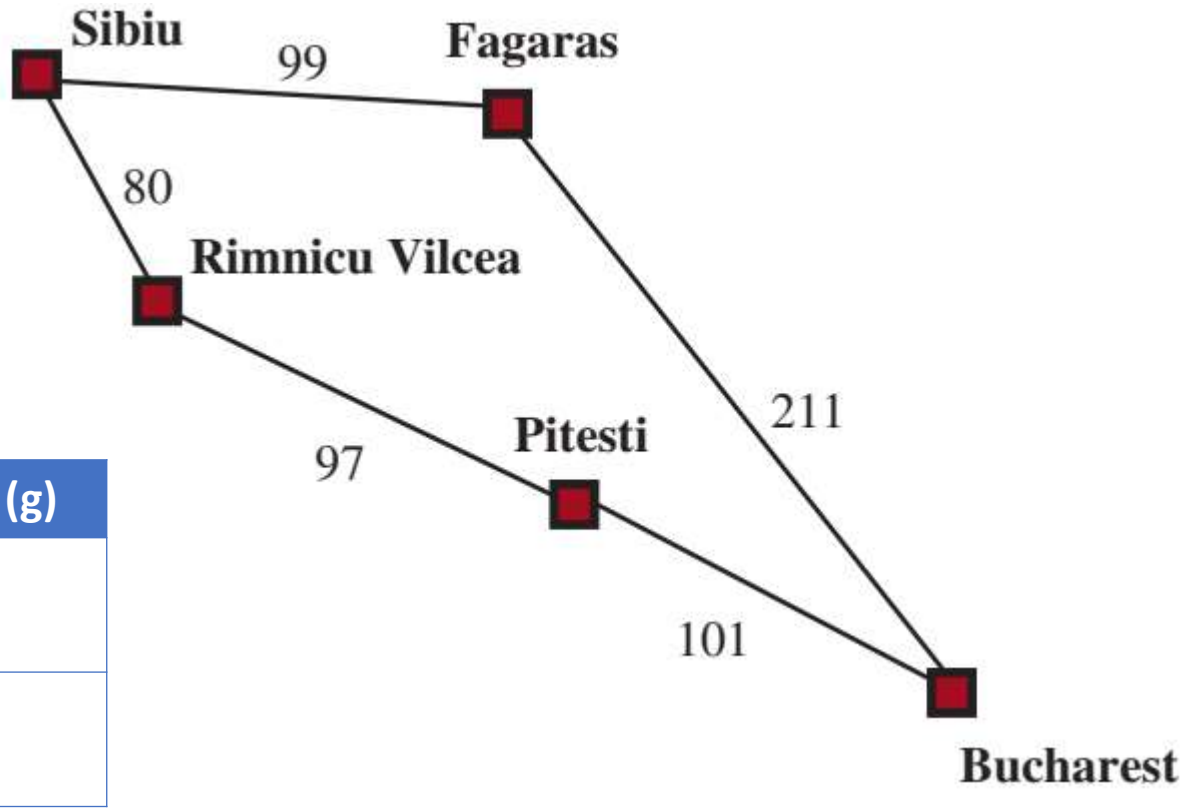
Iterative Deepening Search (IDS)

Dijkstra Search

From “Romania” example

In AIMA

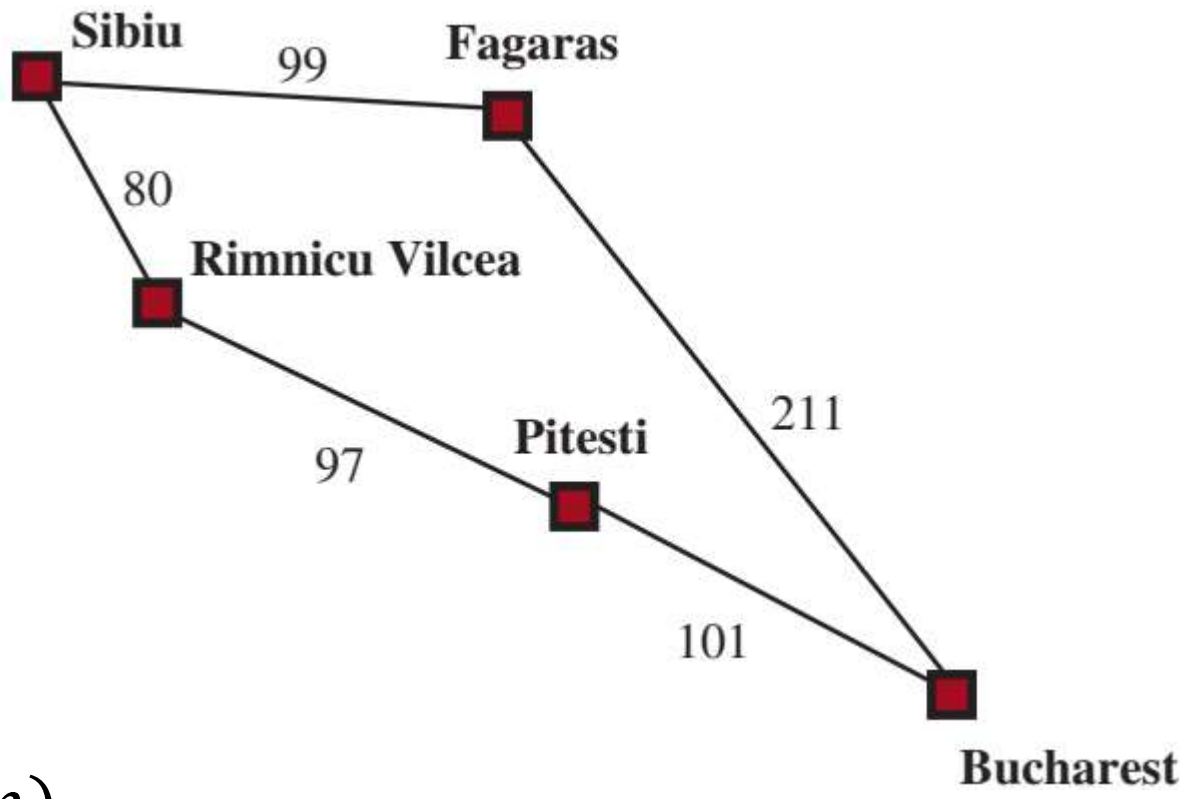
(find path to Bucharest)



Solution	depth	Path cost (g)
Sibiu, Faragas, Bucharest	2	310
Sibiu, R. V., Pitesti, Bucharest	3	278

Dijkstra Search

Node	d	g
Sibiu	0	0

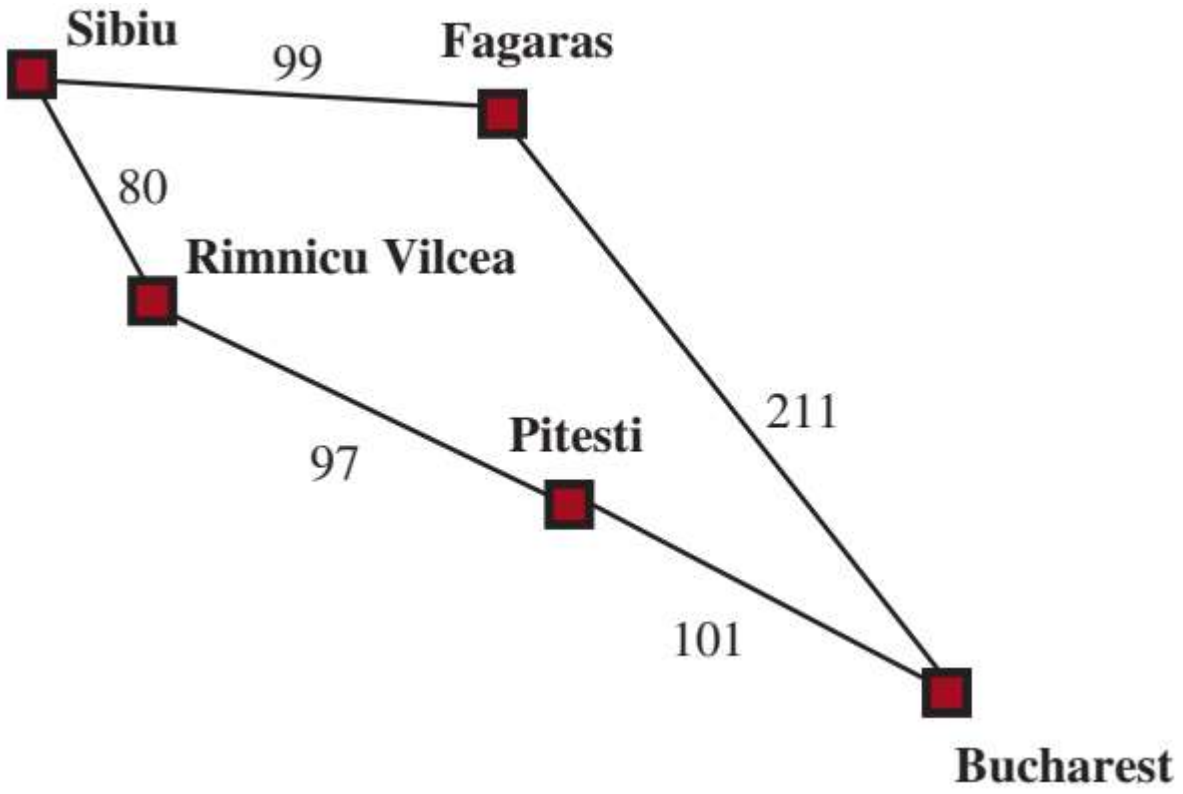


Search strategy:

Queue nodes in order of $g(n)$

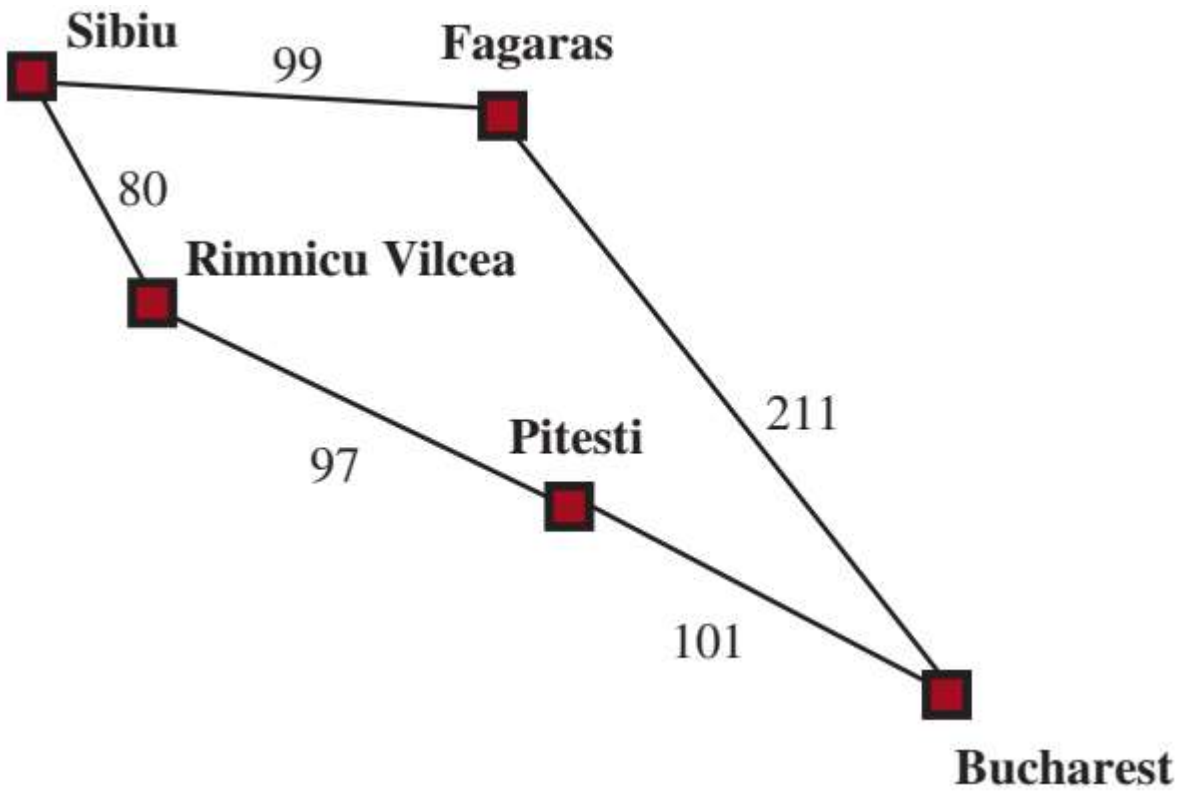
Dijkstra Search

Node	d	g
Rimnicu Vilcea	1	80
Fagaras	1	99



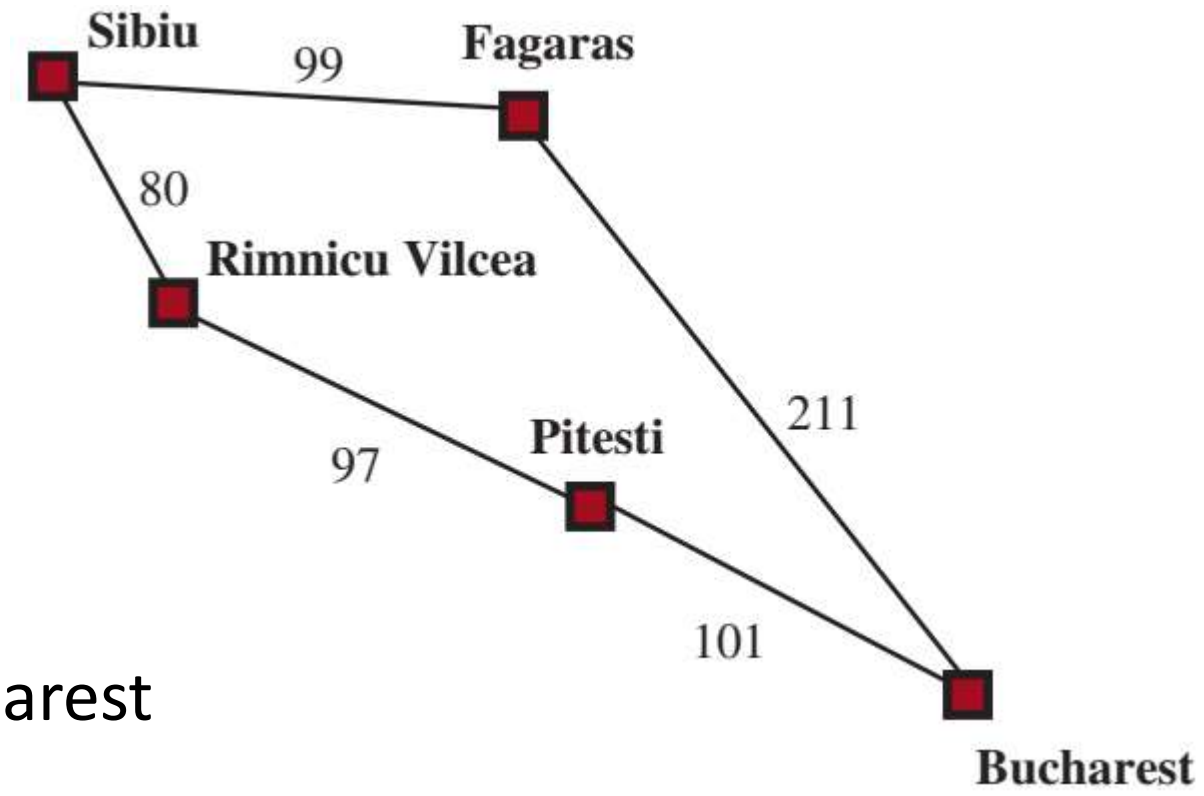
Dijkstra Search

Node	d	g
Fagaras	1	99
Pitesti	2	177



Dijkstra Search

Node	d	g
Pitesti	2	177
Bucharest	2	310

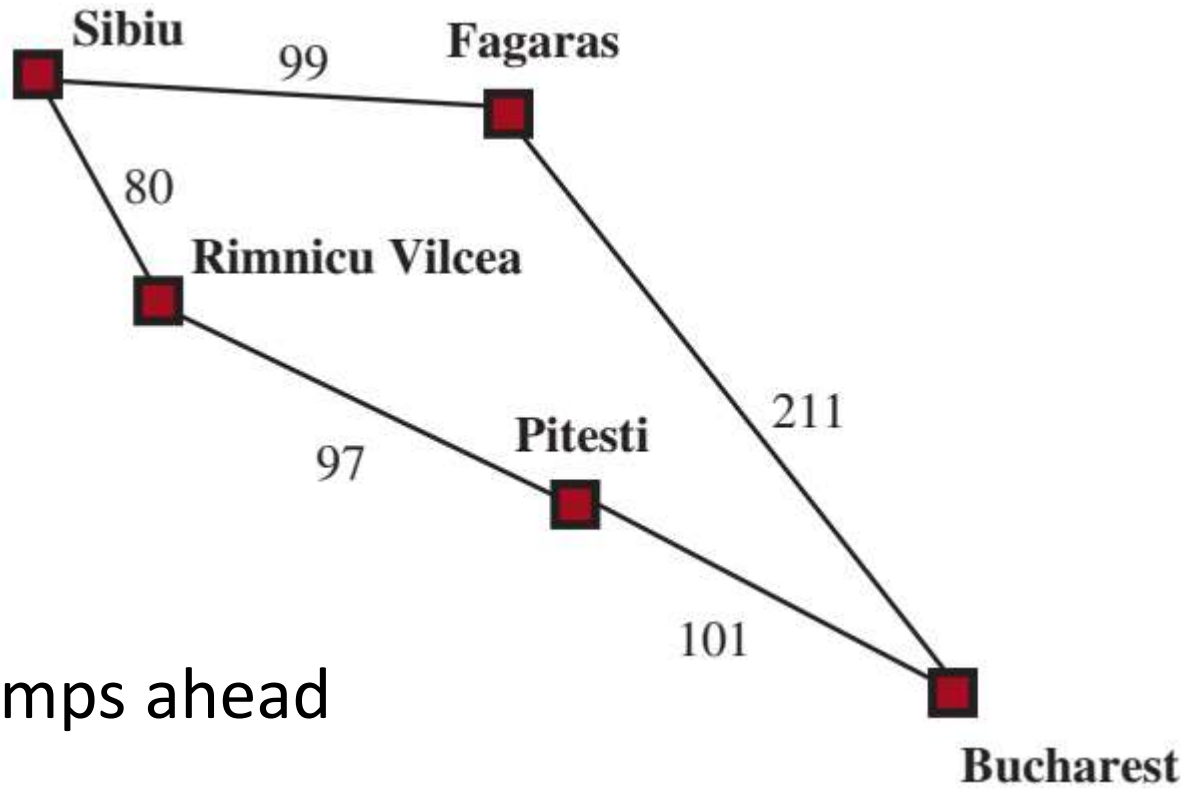


Pitesti expands before Bucharest

(lower cost)

Dijkstra Search

Node	d	g
Bucharest	3	278
Bucharest	2	310



Second path to Bucharest jumps ahead (lower cost)

Dijkstra Search

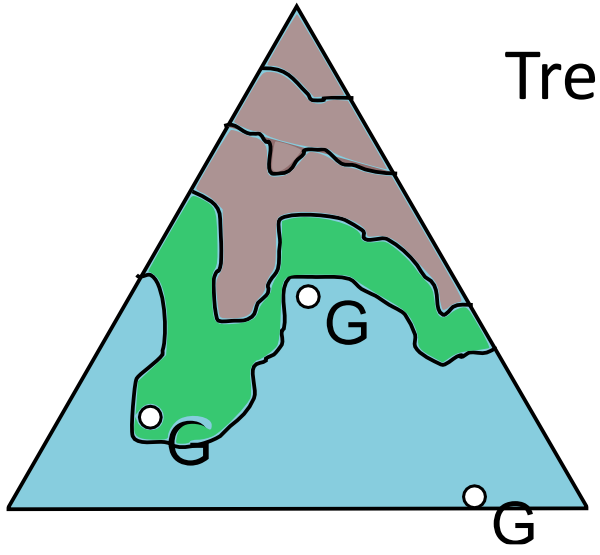
Optimal

(and uses cost!)

“Complete” -

Always finds

finite depth goal



Tree is explored by cost $g(n)$

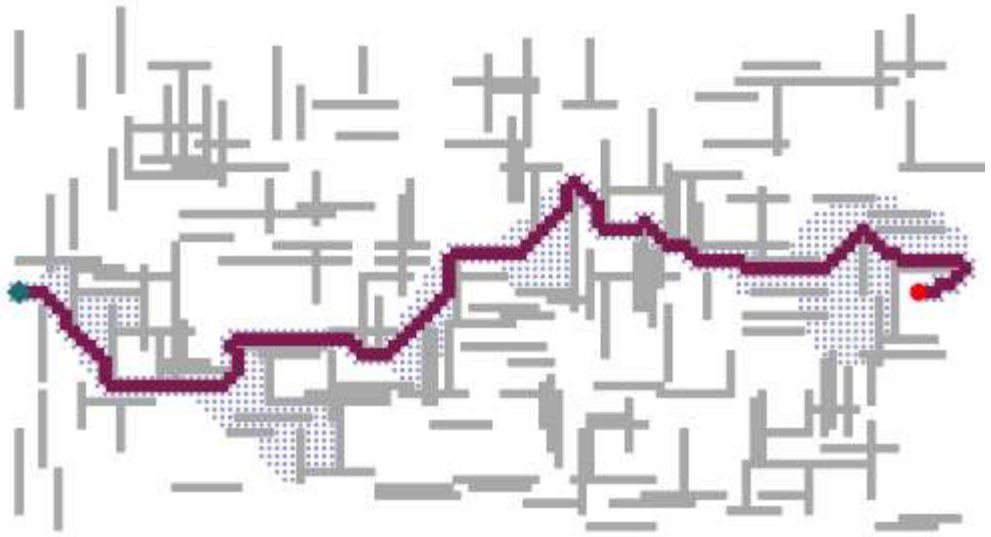
(not number of steps)

Heuristic search

If we know the direction of the goal

why search in the opposite direction?

Heuristic – some function to measure closeness to goal



Heuristic Search

Order search queue by $f(n)$

Strategy	Queue order	Cares about
Dijkstra	$f(n) = g(n)$	Path cost
Greedy	$f(n) = h(n)$	Closeness to goal
A*	$f(n) = g(n) + h(n)$	Both!

$h(n)$-distance from node to goal

Heuristic Search

Heuristic functions:

Recall: SLD, Manhattan for path finding on 2D maps

Exercise:

Find heuristics (how to estimate # of steps to goal) for this game:

(Solutions in AIMA book 3.6)

Heuristic search

$h(n)$ is “admissible” if it never **over**estimates

A* optimality proof:

https://youtu.be/iTJvWfmp1vw?t=208

Strategies summary

Strategy	Optimal for # of steps	Optimal for cost	Complete	Time	Space
BFS	Yes	No	Yes	$O(b^d)$	$O(b^d)$
DFS	No	No	No	$O(bm)$	$O(b^m)$
Dijkstra	Yes(1)	Yes	Yes		
Greedy	No	No	No		
A*	Yes(1)	Yes	Yes		

(1) Assuming cost doesn't matter, $c = 1$ achieves shortest # of steps